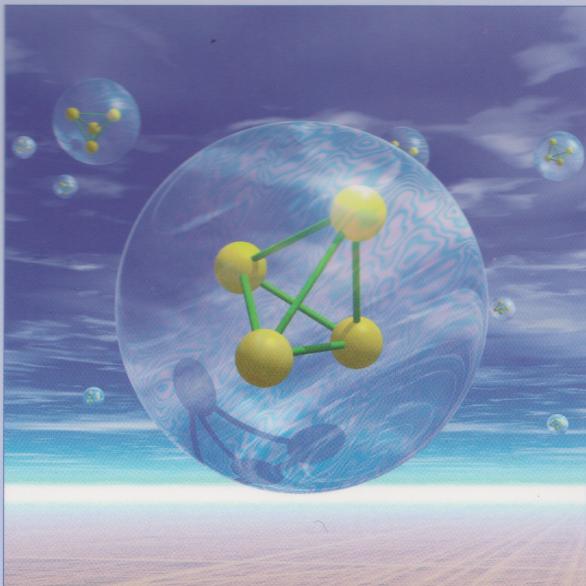




M255 Unit 8

UNDERGRADUATE COMPUTING

Object-oriented programming with Java



Designing code,
dealing with errors

Unit 8



M255 Unit 8
UNDERGRADUATE COMPUTING

Object-oriented programming with Java



**Designing code,
dealing with errors**

Unit 8

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at <http://www.open.ac.uk> where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit <http://www.ouw.co.uk>, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008.

Copyright © 2006, 2008 The Open University.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher or a licence from the Copyright Licensing Agency Ltd. Details of such licences (for reprographic reproduction) may be obtained from the Copyright Licensing Agency Ltd of 90 Tottenham Court Road, London, W1T 4LP.

Open University course materials may also be made available in electronic formats for use by students of the University. All rights, including copyright and related rights and database rights, in electronic course materials and their contents are owned by or licensed to The Open University, or otherwise used by The Open University as permitted by applicable law.

In using electronic course materials and their contents you agree that your use will be solely for the purposes of following an Open University course of study or otherwise as licensed by The Open University or its assigns.

Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or retransmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of The Open University or in accordance with the Copyright, Designs and Patents Act 1988.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 978 0 7492 5500 8

2.1

The paper used in this publication contains pulp sourced from forests independently certified to the Forest Stewardship Council (FSC) principles and criteria. Chain of custody certification allows the pulp from these forests to be tracked to the end use (see www.fsc.org).



CONTENTS

Introduction	5
1 Formatting Java code	6
1.1 Indentation	6
1.2 Self-documenting code	10
1.3 Wrapping long lines of code	11
2 Developing the Converter class	14
3 Errors in programming	27
3.1 Compile-time errors	27
3.2 Run-time errors	32
4 Debugging	42
4.1 Debugging if, for and while statements	42
4.2 Tracing execution	45
5 Summary	49
Glossary	50
Index	52

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Academic Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Associate Lecturer, Author and Critical Reader

Robin Walker, Critical Reader, Associate Lecturer

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

When you write a program you express it as text in a particular programming language, Java in the case of M255. The code you write has to comply with the syntax rules of the language – its grammar – and you may need to correct quite a few syntax errors before what you have written is accepted by the Java compiler.

However, getting the code to compile is only the first step; it is simply static code. To make anything happen, it has to be executed.

During the development of a program our attention is naturally focused on the programming statements we are writing and we do not tend to think much about its execution. In fact we often come to think of what we are writing as though it were the running code, which it is not. When we actually run a program, it is highly likely we shall find things do not go according to plan – the program may work incorrectly or it may even crash. Then we need to debug it by diagnosing and correcting program errors.

In this unit we start by showing good practice in how to format your code in order to make it more readable and understandable. Next we take you through the design of a new class, the class `Converter` – this will serve to reprise and put into practice the ideas you have learnt from earlier units. The second half of this unit looks at errors in code, both compile-time and run-time errors and how to interpret the error messages given by the BlueJ compiler. The unit ends with some techniques for debugging code.

1

Formatting Java code

One of the goals of M255 is for you to learn how to write programs that are not only correct but also understandable.

Your code should be readable, because if it is not readable, the chances of it being correct are slim. Moreover, if your program is unreadable, it will be difficult and time consuming to find and correct any errors in it. Your code should also be readable by others, not just you. Remember some TMA questions ask you to write Java code – if your tutor has difficulty reading and understanding your code how are they going to mark it accurately? Outside this course, making code readable by others becomes even more important. Most programs exist for a long time and require ‘maintenance’ – changes to adapt to new and different requirements, upgrades in other software, new hardware. The author of a program is quite likely not to be around when maintenance is required; someone else must read the program and understand it enough to update it successfully. Even a program you write for yourself should be readable; if not, soon after finishing it you will probably not remember it well enough to make changes easily. Thus, it makes sense to develop programming habits that lend themselves to writing readable, understandable and correct programs.

In this section we introduce you to a number of **formatting guidelines** that will make your code clearer to read, understand and debug.

1.1 Indentation

While the use of braces tells the **compiler** which lines of code belong to a particular code block, such as a class, method, or `if` statement, indentation can be used to make the structure of code clear to a human reader. In the following discussion we talk about indenting by multiples of three spaces rather than tabbing using the tab key of your keyboard. The reason we stipulate this is that generally, in an editor, if you use the tab key to provide indentation then the format of your code will not look the same if you send it to a friend or colleague when they open up the code in an editor which has not got the tabs set to the same number of spaces. However, when you hit the tab key in the BlueJ editor it does not enter the unicode tab character, it enters proper spaces – so using the BlueJ editor you can happily use the tab key and be sure that your code will look the same no matter what editor subsequently reads the code. By default, the tab key in BlueJ enters four spaces which many programmers prefer. However in this course we have set the tab in BlueJ to be three spaces to avoid too much line wrapping. So in the following discussion when we talk about three spaces you can just use the tab key *if* you are using the BlueJ editor.

Classes

When writing a new class, any class comment, any import statement and the class header should not be indented at all – they should be hard left. The matching pair of braces that delimit the class should always appear on their own lines and should line up with each other and not be indented. This tells a human reader the extent of the class. Everything else within a class should then be indented relative to the enclosing class block. The indentation is achieved by entering three spaces. Here is an example.

```
import ou.*;
public class Converter
{
...
}
```

Variables

Instance and static variable declarations should be indented by three spaces relative to the class block, as shown below.

```
import ou.*;
public class Converter
{
    private static double fixedFee = 2.00;
    private static double percentageRate = 0.05;
    private double exchangeRate;
    private String currencyName;
    ...
}
```

Methods

The method header and the braces that delimit the method block should be indented by three spaces. The matching pair of braces that delimit the code within a method should always appear on their own lines and should line up with each other. This tells a human reader the extent of the method. *All* statements within a method should then be indented by (at least) another three spaces so that it is clear to a reader that they form part of a particular method. Here is an example.

```
import ou.*;
public class Converter
{
    private static double fixedFee = 2.00;
    private static double percentageRate = 0.05;
    private double exchangeRate;
    private String currencyName;

    public void foo()
    {
        < method statement block >
    }
    ...
}
```

if statements

Notice that the `if` keyword is indented by three spaces relative to the method's opening and closing braces, and that the opening and closing braces for the `then` statement block are also indented by three spaces and line up vertically with each other. The statements within the `then` statement block are indented by a further three spaces. Notice too that the `else` keyword lines up vertically with the `if` keyword, and the statements within the `else` statement block are indented by a further three spaces. See the following example.

Remember that `then` isn't actually a Java keyword but we will leave it in code styling when referring to a statement block, to indicate that we are talking about a section of code.

```

public void foo()
{
    if (<condition>)
    {
        < then statement block >
    }
    else
    {
        < else statement block >
    }
}

```

while statements

The `while` keyword is also indented by three spaces and the opening and closing braces for the statements in the `while` statement block are also indented by three spaces and line up vertically with each other. The statements in the `while` statement block are indented by a further three spaces.

```

public void foo()
{
    while (<condition>)
    {
        < statement block >
    }
}

```

for statements

As you can see in the example below, the indentation of the `for` statements are the same as for the `while` statements.

```

public void foo()
{
    for (<init>; <test>; <inc>)
    {
        < statement block >
    }
}

```

Nested statements

Indentation within nested statement blocks is a little more complex, as in the following example of an `if` statement nested within a `while` loop.

```

public void foo()
{
    while (<condition>)
    {
        if (<condition>)
        {
            < statement block >
        }
    }
}

```

Note that the indentation of the `if` statement is *relative* to the `while` loop. Here is an even more complicated example where a `for` loop is nested within an `if` statement block.

```
public void foo()
{
    while (<condition>)
    {
        if (<condition>)
        {
            for (<init>; <test>; <inc>)
            {
                < statement block >
            }
        }
    }
}
```

Here the `for` loop is indented relative to the `if` statement which in turn is indented relative to the `while` loop. By indenting the code in this manner it makes it clear that the `for` loop is inside the `if` statement block and that the `if` statement is inside the `while` statement block – a bit like Russian dolls.

We have stated throughout this section that the matching pair of braces that delimit blocks of code should always appear on their own lines and should line up with each other. However there is another convention that is popular which you might see in programming books and it looks like this:

```
public void foo() {
    if (<condition>) {
        < then statement block >
    }
    else {
        < else statement block >
    }
}
```

(Note that this is the same as the code example that appears under the 'if statement' heading.) This style of coding is popular because it takes up fewer lines. However, on this course we prefer to have pairs of braces on their own lines as we believe it makes the code clearer to read.

Spaces in code

There are also some stylistic guidelines about the use of spaces in Java. Thus, `if`, `for` and `while` statements should have a space before the opening parenthesis of the condition, for example:

```
while (<condition>)
```

not

```
while(<condition>)
```

Method headers should *not* have a space before the opening parenthesis, because it helps us to distinguish method names from keywords such as `while`. So we write:

public void foo()
not

public void foo ()

There should be a space after each comma in a method's argument list so that we can clearly identify the individual arguments:

public void foo(a, b, c)
not

public void foo(a,b,c)

Everything we have said above about spacing in method headers is also true of message-sends, so use:

this.foo(a, b, c)
not

this.foo (a,b,c)

Finally there should be a space before and after an operator, for example, we write:

x + y
not

x+y

The dot operator is an exception, we always write:

this.foo()
not
this . foo()

1.2 Self-documenting code

Rather than trying to write inline comments to explain how a method performs a complex task, try to make the code easier to read by introducing local variables which serve to label intermediate steps in the logic of a method.

Consider this method which determines whether or not a particular year is a leap year:

```
/**  
 * Return true if the year given by the argument (a four digit  
 * integer) is a leap year, false otherwise.  
 *  
 * Leap years occur in years divisible by 4, except that years  
 * ending in 00 (exactly divisible by 100) are leap years only  
 * if they are exactly divisible by 400.  
 */  
public boolean isLeapYear(int y)  
{  
    return (((y % 400) == 0) || (((y % 4) == 0) && ((y % 100) != 0)));  
}
```

It would take a very experienced eye to understand what this method is doing. In comparison the following version of the method is made much easier to understand by assigning the results of intermediate expressions to meaningfully named local variables.

```
public boolean isLeapYear(int year)
{
    boolean potentialLeapYear = ((year % 4) == 0);
    boolean nonCenturyYear = ((year % 100) != 0);
    boolean nonCenturyLeapYear = (potentialLeapYear && nonCenturyYear);
    boolean centuryLeapYear = ((year % 400) == 0);
    return nonCenturyLeapYear || centuryLeapYear;
}
```

Another approach to writing this method with local variables would be:

```
public boolean isLeapYear(int year)
{
    boolean divisibleBy4 = ((year % 4) == 0);
    boolean divisibleBy100 = ((year % 100) == 0);
    boolean divisibleBy400 = ((year % 400) == 0);
    return (divisibleBy400 || (divisibleBy4 && ! divisibleBy100));
}
```

The declaration and use of the local variables make no difference to the efficiency or speed of the compiled code but make a great deal of difference to the efficiency and speed of a human reader!

1.3 Wrapping long lines of code

When an expression will not fit on a single line, break it according to these general principles:

- ▶ break after a comma;
- ▶ break before an operator;
- ▶ align the new line with the beginning of the sub-expression at the same level on the previous line.

Here are two examples of putting line breaks in the argument list of message-sends:

```
obj1.someMessage(argument1, argument2, argument3,
                  argument4, argument5);

someVar = obj2.someMessage1(argument1,
                            obj3.someMessage2(argumentA, argumentB));
```

Here is an example of a `println()` message where the argument is made up of a number of concatenated strings:

```
System.out.println("This is a rectangle with length " + this.length()
                  + "breadth " + this.breadth()
                  + " Its area is " + this.getArea()
                  + " Its perimeter is " + this.getPerimeter() + ".");
```

The following are two examples of breaking an arithmetic expression which is part of an assignment statement. The first example is preferred, since the break occurs after the parenthesised sub-expression.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
           + 4 * longname6; // PREFER
```

```
longName1 = longName2 * (longName3 + longName4
           - longName5) + 4 * longname6;
```

The second example of line breaking should be avoided as the break is within the sub-expression `(longName3 + longName4 - longName5)` so making that sub-expression harder to read.

Another suitable place to put a line break is after the `=` sign:

```
int age =
    Integer.parseInt(OUDialog.request("Please enter your age"));
```

Exercise 1

Using pencil and paper, format the following code (chosen at random from a Java training site on the Web!) along the lines of the guidelines given in this section. You do not need to try and understand what the code does – just apply the guidelines.

```
private Vector inspectCollection( Object obj ){
    Vector dataV = new Vector();
    Vector rowV = new Vector();
    rowV. add( "size" );
    rowV .add("int");
    rowV.add( Integer. toString(((Collection)obj).size()) );
    dataV.add(rowV);
    rowV = new Vector();
    Iterator it = ((Collection) obj).iterator();
    Object anObj;
    int i = 0;
    while (it.hasNext()) {
        anObj = it.next();
        rowV.add("[ "+Integer.toString( i )+
        "]");
        rowV.add(anObj.getClass().getName());
        if (checkForPrimitive(anObj))
        {rowV.add(anObj.toString());}
        else
        {
            if (anObj instanceof java.lang.String)
            {rowV.add(anObj);}
            else
            {rowV.add("Inspect object");}
            hm.put("[ "+Integer.toString(i)+" ]", anObj);
        }
        dataV.add(rowV);
        rowV = new Vector();
        i++;
    }
    return dataV;
}
```

Solution

```
private Vector inspectCollection(Object obj)
{
    Vector dataV = new Vector();
    Vector rowV = new Vector();
    rowV.add("size");
    rowV.add("int");
    rowV.add(Integer.toString(((Collection) obj).size()));
    dataV.add(rowV);
    rowV = new Vector();
    Iterator it = ((Collection) obj).iterator();
    Object anObj;
    int i = 0;
    while (it.hasNext())
    {
        anObj = it.next();
        rowV.add("[" + Integer.toString(i) + "]");
        rowV.add(anObj.getClass().getName());
        if (checkForPrimitive(anObj))
        {
            rowV.add(anObj.toString());
        }
        else
        {
            if (anObj instanceof java.lang.String)
            {
                rowV.add(anObj);
            }
            else
            {
                rowV.add("Inspect object");
            }
            hm.put("[" + Integer.toString(i) + "]", anObj);
        }
        dataV.add(rowV);
        rowV = new Vector();
        i++;
    }
    return dataV;
}
```

SAQ 1

Why is it important to format code in accordance with a set of commonly accepted guidelines?

ANSWER

The most important reason is that it will make your code more understandable to a reader. This makes it easier to debug and maintain. There is something more: to format code clearly, you need to understand it clearly, and so the process of formatting it is part of being sure you have got it right. Also when code is delivered to a customer they will expect to see consistent formatting and this is best arranged by everyone using the same rules.

2

Developing the Converter class

In this section we are going to investigate how to design and implement a solution to a particular programming problem.

Walton Bureau de Change is a small company based in Milton Keynes that changes pounds sterling into the various foreign currencies needed by Open University staff when they travel abroad on business. Walton Bureau de Change only sell foreign currencies; they do not buy back any unused foreign currency. Also, they only sell whole numbers of a foreign currency, i.e. they will sell \$150, but not \$150.45 as they do not keep foreign currency small change. For this service Walton Bureau de Change charge a commission, and this commission consists of two fees: a fixed fee for handling the transaction, plus a percentage fee that is related to the cost (in pounds) of the foreign currency being purchased.

An example will clarify how the costs of buying foreign currency from Walton Bureau de Change are worked out.

Suppose we want to buy \$150. Also suppose that the exchange rate is \$1.5 to £1 sterling, that the transaction fixed fee is £2.00 and that the percentage fee is 5% of the number of dollars being purchased. Then the percentage component of the commission, in dollars, is given by:

$$5\% \text{ of } \$150 = 0.05 * 150 = \$7.50.$$

This percentage fee part of the commission can be converted into sterling:

$$7.5/1.5 = £5.00.$$

The total commission is the sum of the fixed fee and the percentage fee, that is:

$$£2.00 + £5.00 = £7.00.$$

The exchange price of \$150 in £ sterling is $150 / 1.5 = £100$, and so the total cost of purchasing \$150 from Walton Bureau de Change is:

$$£7 + £100 = £107.$$

We can generalise these calculations in the following descriptive mathematical formulas.

$$\text{percentage fee} = (\text{percentage rate} * \text{amount of currency}) / \text{exchange rate}$$

$$\text{commission} = \text{fixed fee} + \text{percentage fee}$$

$$\text{total cost} = \text{commission} + (\text{amount of currency} / \text{exchange rate})$$

At the moment Walton Bureau de Change work out these calculations by hand, using pencil and paper which by its very nature is error prone. In fact Walton Bureau de Change have had a number of complaints recently from customers who have been overcharged. Therefore Walton Bureau de Change have asked us to come up with a software solution that will do these calculations for them.

Exercise 2

A software solution for Walton Bureau de Change can be achieved by writing a single class. Suggest the name of that class and from what class it should be subclassed.

Solution.....
Instances of this new class will be used to calculate the cost of buying various foreign currencies in pounds sterling, so for example one instance of the class might be used to calculate the cost of buying US dollars and another may be used to calculate the cost of buying euros. A suitable name for the class, that reflects what instances of the class actually do, would be something like `ForeignCurrencyCalculator`, `CurrencyTransactionCalculator`, `CurrencyConverter`, or just simply `Converter`. For brevity we shall chose the name `Converter`.

So far in the course you have not come across a class that is similar to this new class, therefore the class should be created as a subclass of `Object`.

Exercise 3

The class `Converter` will be created as a subclass of `Object`, and your next task is decide what instance and class (static) variables will be needed.

To decide what variables are needed you need to work out what attribute values instances of the class will need to remember in order to carry out their behaviour, i.e. to calculate the cost of a particular currency and report that cost to the user. To help you decide, here is a description of what an instance of the class must do.

Instances of the `Converter` class are to be used to calculate the cost of buying a particular/different foreign currency in pounds sterling, so for example one instance of the class might be used to calculate the cost of buying US dollars and another may be used to calculate the cost of buying euros. Instances of the class will need different exchange rates in order to correctly calculate the cost of buying a particular foreign currency. The commission for all transactions (regardless of the particular currency an instance of the class has been created to work with) will be calculated using the same fixed fee and the same percentage rate. Once an instance of the class has calculated the cost of buying the currency it should report that cost to the user via a dialogue box.

When deciding which class variables will be needed, you need to ask yourself what attributes will have the same values for all instances of the class.

When deciding which instance variables will be needed, you need to ask yourself what attributes will probably have different values for each instance of the class.

List what instance variables and class (static) variables will be needed to implement the class. State also what types these class and instance variables should be declared as.

Solution.....

The class will need the following class variables:

- ▶ `fixedFee` – to hold the fixed part of the commission. This should be declared as a double, because the fixed part of the commission needs to be a decimal number that represents a value in £ sterling.
- ▶ `percentageRate` – to hold the percentage to be used to calculate the percentage part of the commission. This should be declared as a double, because the percentage rate is to be represented as a decimal number.

These variables need to be class variables, because every instance of the class will need to calculate the commission cost based on the same values for `fixedFee` and `percentageRate`.

The class will need the following instance variables:

- ▶ `exchangeRate` – to hold the exchange rate for a particular currency. This should be declared as a `double`, because the exchange rate is to be represented as a decimal number.
- ▶ `currencyName` – to hold the name of the currency. This should be declared as a `String`, because currency names will be strings such as "US dollars" or "euros". The value of this instance variable will be useful when reporting the cost of buying an amount of a particular currency in a dialogue box.

`exchangeRate` and `currencyName` need to be instance variables as their values will differ for each `Converter` object.

You may have chosen slightly different names for the class and instance variables, but so long as the names you have chosen clearly indicate the purpose of each variable this is fine and normal.

You may have decided that an instance variable is needed to record the amount of foreign currency to convert for a particular transaction – this is not needed as instances of the class `Converter` do not need to remember the details of the amount of currency to convert for a particular transaction as will become clear once you do Activity 4.

ACTIVITY 1

Launch BlueJ and open the project `Unit8_Project_1`. Double-click on the icon for the `Converter` class. Once the editor has opened write the Java declarations for the class and instance variables after the opening curly bracket, `{`. You should give any class variables default values in your variable declarations.

Given that we will later provide accessor methods for these variables you must decide whether the variables should be declared as `public` or `private`.

DISCUSSION OF ACTIVITY 1

You should have added the following code to the class:

```
private static double fixedFee = 2.00;  
private static double percentageRate = 0.05;  
private double exchangeRate;  
private String currencyName;
```

All the variables should be declared as `private` to implement data hiding. Access to the class and instance variables will be via accessor methods.

Now that we have declared the class and instance variables we also need accessor pairs for them, namely `setExchangeRate()`, `getExchangeRate()`, `setCurrencyName()`, `getCurrencyName()`, `setFixedFee()`, `getFixedFee()`, `setPercentageRate()` and `getPercentageRate()`.

SAQ 2

Explain the purpose of accessor methods using examples.

ANSWER.....

Accessor methods are used to access the values held by the instance variables of an object. Usually each instance and class (static) variable has a get and set method that make up a pair. Naming is usually common to the methods in the pair and the instance variable. For example, the `getExchangeRate()` method returns the value of the receiver's instance variable `exchangeRate`. The `setExchangeRate()` method enables the receiver's instance variable `exchangeRate` to be set to the value of the method's argument.

ACTIVITY 2

If it is not already open, launch BlueJ and open the project `Unit8_Project_2`. Double-click on the icon for the `Converter` class to open the editor; you will see that the instance and class variables from the previous activity have been added to the class file for you.

You are now going to write the accessor methods for the class and instance variables of `Converter`. You must decide the return types and argument types, whether the methods should be declared as `public` or `private` and which should be declared as `static`.

DISCUSSION OF ACTIVITY 2

```
// class methods
/**
 * Sets the fixedFee of the receiver to the value of the argument
 */
public static void setFixedFee(double fee)
{
    Converter.fixedFee = fee;
}
/**
 * Returns the value of the fixedFee of the receiver
 */
public static double getFixedFee()
{
    return Converter.fixedFee;
}
/**
 * Sets the percentageRate of the receiver to the value of the argument
 */
public static void setPercentageRate(double percentage)
{
    Converter.percentageRate = percentage;
}
/**
 * Returns the value of the percentageRate of the receiver
 */
public static double getPercentageRate()
{
    return Converter.percentageRate;
}
```

```

//instance methods
/**
 * Sets the exchangeRate of the receiver to the value of the argument
 */
public void setExchangeRate(double rate)
{
    this.exchangeRate = rate;
}

/**
 * Returns the value of the exchangeRate of the receiver
 */
public double getExchangeRate()
{
    return this.exchangeRate;
}

/**
 * Sets the currencyName of the receiver to the value of the argument
 */
public void setCurrencyName(String nameOfCurrency)
{
    this.currencyName = nameOfCurrency;
}

/**
 * Returns the value of the currencyName of the receiver
 */
public String getCurrencyName()
{
    return this.currencyName;
}

```

Note that all the methods have been declared as `public` because all the static and instance variables have been declared as `private`, therefore the user(s) of the class `Converter` will need to use the accessor methods to get and set the variables.

Note also how within the accessor methods we can differentiate between instance variables and class variables. Instance variables are prefixed by `this`, and class variables are prefixed by the class name. For example:

`this.exchangeRate` denotes the `exchangeRate` *instance* variable of whatever `Converter` object is referenced by `this` at run-time.

`Converter.percentageRate` denotes the `percentageRate` *class* variable.

Qualifying a class variable with the class name is not obligatory within a class's own methods, nor is prefixing an instance variable with `this`. However it does make your code much clearer to read.

Next we need a constructor (other than the default) for objects of the class `Converter`, but what values should the instance variables be set to by this constructor? We might decide to set the instance variables to some default values, for example `exchangeRate` to 1.5 and `currency` to "US dollars" on the basis that often we want a `Converter` object to convert US dollars into sterling. But another user of the class may want to convert euros into sterling more often. So setting the instance variables to default values is not the solution. What we want is a constructor with arguments, the values of which will be used by the constructor to set the instance variables.

ACTIVITY 3

If it is not already open, launch BlueJ and open the project Unit8_Project_3. Double-click on the icon for the `Converter` class to open the editor; you will see that the accessor methods from the previous activity have been added to the class file for you. Now write a constructor for the class `Converter` that takes two arguments: a `String` argument named `nameOfCurrency` and a `double` argument named `rate`. Write your code just after the declaration of the static and instance variables.

DISCUSSION OF ACTIVITY 3

```
public Converter(String nameOfCurrency, double rate)
{
    this.currencyName = nameOfCurrency;
    this.exchangeRate = rate;
}
```

Note that in the constructor we directly assign the values of the arguments to the instance variables rather than use the setter methods. The main reason for this is that if the constructor sends a message whose corresponding method is overridden in a subclass, then problems will arise if that method then tries to use some uninitialised variable.

Now we have the class and instance variables, their accessor methods and a constructor. However a method to carry out the business of converting currency is still missing. So to finish the class `Converter` we need one more method, `costOfCurrency()`. In the next activity you will write the `costOfCurrency()` method which will calculate the cost in pounds of buying an amount of foreign currency.

ACTIVITY 4

If it is not already open, launch BlueJ and open the project Unit8_Project_4. Double-click on the icon for the `Converter` class to open the editor; you will see that the constructor from the previous activity has been added to the class file for you.

In this activity you are going to write the `costOfCurrency()` method. You should write the method at the bottom of the class `Converter`, just before the closing brace. The method takes an `int` as an argument which represents the amount of currency to buy. The method does not return a value but instead uses an appropriate dialogue box to report to the user the total cost of buying the currency in £ sterling

When writing the method remember to use accessor methods (it is not good practice to directly access the instance and class variables). A comment should be included to describe briefly the behaviour caused by the method.

At the beginning of this subsection we generalised the calculations needed to carry out a currency transaction with the following descriptive mathematical formulas which should help you to write the method.

$$\text{percentage fee} = (\text{percentage rate} * \text{amount of currency}) / \text{exchange rate}$$

$$\text{commission} = \text{fixed fee} + \text{percentage fee}$$

$$\text{total cost} = \text{commission} + (\text{amount of currency} / \text{exchange rate})$$

The method should have the following header:

```
public void costOfCurrency(int amountOfCurrency)
```

Once you have calculated the total cost of the transaction you will need to display the result in a dialogue box. To do this you will need to concatenate strings and numbers together to create a suitable string argument for an `alert()` dialogue box. If the exchange rate is, for example, 1.7, the message displayed will be:

150 dollars cost £94.6470588235294

Obviously, this is far too many decimal places, so, if you are feeling confident and looking for an extra challenge, you can round the final cost of the transaction to two decimal places. To do this, you will need first to import the `DecimalFormat` class at the top of the `Converter` class file, with the statement:

```
import java.text.DecimalFormat;
```

This is a class supplied by Java to format string representations of decimal numbers. Once you have done that, in the `costOfCurrency()` method write code to create an object of this class and assign it to a variable as follows:

```
DecimalFormat df = new DecimalFormat("0.##");
```

Then, when you need to display the cost of the currency, the message expression

```
df.format(cost)
```

will ensure that the string representation of `cost` will be displayed as a decimal number with only two decimal places.

Ensure that the `Converter` class compiles correctly before moving on.

DISCUSSION OF ACTIVITY 4

```
/*
 * Calculate and report cost in sterling of buying an amountOfCurrency
 */
public void costOfCurrency(int amountOfCurrency)
{
    double percentageFee, commission, cost;
    DecimalFormat df = new DecimalFormat("0.##");
    percentageFee = Converter.getPercentageRate() * amountOfCurrency
        / this.getExchangeRate();
    commission = Converter.getFixedFee() + percentageFee;
    cost = commission + (amountOfCurrency / this.getExchangeRate());
    OUDialog.alert(amountOfCurrency + " " + this.getCurrencyName()
        + " cost £" + df.format(cost));
}
```

Note how within the `costOfCurrency()` method we can differentiate between message-sends to objects and invocations of a class's static methods.

- ▶ `this.getExchangeRate()` is a message-send. The compiler cannot determine the corresponding method code for `getExchangeRate()`. The message `getExchangeRate()` will be sent to whatever object is referenced by `this` at run-time, which of course could be an object of some subclass of `Converter` that has overridden the `getExchangeRate()` method declared in the class `Converter`.
- ▶ `Converter.getPercentageRate()` is not a message-send; it is an invocation of a *static* method. The code for that method can be determined at compile time as the code `Converter.getPercentageRate()` explicitly tells the compiler that the method can be found in the `Converter` class.

SAQ 3

Explain the use of the local variables `percentageFee`, `commission` and `cost` in the method `costOfCurrency()`. Will these variables be available outside the method?

ANSWER.....

The values which the local variables `percentageFee`, `commission` and `cost` hold are only 'remembered' by those variables for the lifetime of the execution of the method `costOfCurrency()`. These local variables cannot be accessed from outside the method and once `costOfCurrency()` has finished executing, the variables no longer have any meaning. More generally, local variables only exist for the lifetime of execution of the method in which they are declared.

The term **scope** describes the areas of program code from which a variable may be used. We say that the scope of a local variable is the statement block in which it is declared and any nested statement block.

ACTIVITY 5

If it is not already open, launch BlueJ and open the project `Unit8_Project_5`. The class `Converter` in this project has had the code from the previous activity added. (You can if you wish use `Unit8_Project_4` to which *you* added the `costOfCurrency()` method if you got it to compile successfully.)

Open the OUWorkspace and convince yourself that an object of the class `Converter` works as described above by executing the following statements (one line at a time) in the OUWorkspace.

Now create a `Converter` object:

```
Converter dollarConverter = new Converter("US dollars", 1.7);
```

Now try some currency transactions:

```
dollarConverter.costOfCurrency(300);
dollarConverter.costOfCurrency(250);
```

In this unit differing values for the exchange rate have been used deliberately to mirror the fluctuations encountered in real life.

The next statement will cause an exception. Can you think why?

```
dollarConverter.costOfCurrency(2.6);
```

Change the value of the object's `exchangeRate` instance variable:

```
dollarConverter.setExchangeRate(2.0);
```

Change the value of the class variable `percentageRate`:

```
Converter.setPercentageRate(0.07);
```

Now try two more currency conversions:

```
dollarConverter.costOfCurrency(300);
dollarConverter.costOfCurrency(250);
```

DISCUSSION OF ACTIVITY 5

`Converter.costOfCurrency(300);` will display a dialogue box with the message:

300 US dollars cost £187.29

`dollarConverter.costOfCurrency(250);` will display a dialogue box with the message:

250 US dollars cost £156.41

`dollarConverter.costOfCurrency(2.6);` results in OUWorkspace reporting an error message in the display pane:

Semantic error: Message `costOfCurrency(double)` not understood by class 'Converter'

The reason for this is that the method `costOfCurrency()` has the following signature:

`costOfCurrency(int)`

but we have sent a `costOfCurrency()` message with a `double` value as the argument. What this error message is telling us is that there is no method defined for `Converter` objects called `costOfCurrency()` that takes a `double` as its argument. The reason that we wrote the method to take an `int` argument is because Walton Bureau de Change sell only whole numbers of a foreign currency, i.e. they sell \$150, but not \$150.45 as they do not keep foreign currency small change.

After changing the values of the `exchangeRate` instance variable and the `percentageRate` class variable we have the following.

- ▶ `dollarConverter.costOfCurrency(300);` displays a dialogue box with the message:

300 US dollars cost £162.5

- ▶ `dollarConverter.costOfCurrency(250);` displays a dialogue box with the message:

250 US dollars cost £135.75

Walton Bureau de Change, although initially happy with our solution to their problem, have decided that they want some additional functionality; their staff have been complaining that the new system requires them to do too much typing. Often they might have to do a number of transactions for the same currency one after the other; they want to be able to send a message to an initialised `Converter` object which will ask them to enter the amount of currency needed, display the cost, and then ask if they would like another transaction. If they click 'Yes', they will be asked for the amount of currency and again the cost will be displayed. They want this to repeat until, when prompted for another transaction, they click 'No'.

ACTIVITY 6

If it is not already open, launch BlueJ and open the project `Unit8_Project_6`. The class `Converter` in this project has had the code from the previous activities added.

In this activity you are going to add another instance method to the `Converter` class – the header of the method will be as follows:

```
public void doTransactions()
```

The new method does not take an argument. It prompts the user to enter the amount of currency via a `request()` dialogue box. Note that whatever the user inputs into the `request()` dialogue box will have to be converted into an integer using the `parseInt()` method you encountered in *Unit 5*. For example, if the `String "150"` was referenced by the `variable myString` and you wanted to convert it to its `int` equivalent and assign the result to a local `variable myInt`, you could do it as follows:

```
int myInt = Integer.parseInt(myString);
```

`myInt` would then hold the `int` value 150.

Once the cost of the currency in sterling has been calculated and displayed by a `costOfCurrency()` message the method asks the user via a `confirm()` dialogue box if they wish to calculate the cost of another transaction. The method will continue to ask the user (after the cost of a transaction has been calculated) if they wish to calculate the cost of another transaction until the user clicks the No button of the `confirm()` dialogue box.

To write this method you will need a loop of some sort; should it be a `while` loop or a `for` loop? You decide, but if you are not sure, ask yourself: can you predict the number of times the loop will need to execute? Once you have written the method, and it has compiled without errors, open the OUWorkspace, create a new instance of `Converter` that will calculate the cost of buying euros and then test your method by sending a `doTransactions()` message to that new instance. Note that if the string you enter into the `request()` dialogue box cannot be translated into an `int` argument by the method `parseInt()` a run-time error will occur (see Section 3).

DISCUSSION OF ACTIVITY 6

Your code should look similar to the following:

```
public void doTransactions()
{
    String amountString;
    int amountOfCurrency;
    boolean repeat = true;
    while (repeat)
    {
        amountString = OUDialog.request("How many "
            + this.getCurrencyName() + "?");
        amountOfCurrency = Integer.parseInt(amountString);
        this.costOfCurrency(amountOfCurrency);
        repeat = OUDialog.confirm("Would you like another transaction?");
    }
}
```

Because we cannot determine in advance how many times the loop must execute (the user may want one, five or even 20 transactions to be calculated) a `while` loop is the most natural choice. (The method could be coded using a form of `for` loop that you have not encountered and it will not be used in this course.)

```
public void doTransactions()
{
    String currencyString;
    int amountOfCurrency;
    for (boolean repeat = true; repeat;)
    {
        amountString = OUDialog.request("How many "
            + this.getCurrencyName() + "?");
        amountOfCurrency = Integer.parseInt(amountString);
        this.costOfCurrency(amountOfCurrency);
        repeat = OUDialog.confirm("Would you like another transaction?");
    }
}
```

Using a `for` loop in this manner is considered bad practice as a `for` loop should only be used for writing code that is to be repeated a predetermined, fixed number of times.

To test your method you should have written and executed something like the following in the OUWorkspace:

```
Converter euroConverter = new Converter("euros", 1.45);
euroConverter.doTransactions();
```

A working version of the class Converter can be found in Unit8_Project_7.

Walton Bureau de Change are delighted with the new functionality, but a couple of weeks later they come back and tell us:

A lot of prospective customers are wanting to know how much commission they will have to pay before buying a certain amount of a particular currency; the software you have provided does not allow us to display this information and so either the customer walks away or a member of staff has to work this out with pen and paper – this is most unsatisfactory – please add this additional functionality to the software.

In the next activity you will do precisely that!

ACTIVITY 7

In this activity you must do three things:

- ▶ write a method called `calculateCommission()` which will calculate the total commission payable if one were to purchase a certain amount of a particular currency;
- ▶ write a `displayCommission()` method that makes use of the `calculateCommission()` method to display the commission cost;
- ▶ modify the `costOfCurrency()` method so that it does not duplicate the code in your `calculateCommission()` method; instead `costOfCurrency()` should send a `calculateCommission()` message to `this` in order to get the cost of commission.

If it is not already open, launch BlueJ and open the project Unit8_Project_7. The class Converter in this project has had the code from the previous activity added.

Write the method `calculateCommission()`, its purpose is to return the total commission charge for the transaction. The method will take an `int` argument which represents the amount of currency and it will return a value of type `double` which will be the commission cost. Here is the method header:

```
public double calculateCommission(int amountOfCurrency)
```

You already know how to calculate the commission for a particular transaction, you did it in the `costOfCurrency()` method you wrote in Activity 4. The code below in bold is the code that calculates the commission and it can be adapted for use in this new method.

```
public void costOfCurrency(int amountOfCurrency)
{
    double percentageFee, commission, cost;
    DecimalFormat df = new DecimalFormat("0.##");
    percentageFee = Converter.getPercentageRate() * amountOfCurrency
        / this.getExchangeRate();
    commission = Converter.getFixedFee() + percentageFee;
    cost = commission + (amountOfCurrency / this.getExchangeRate());
    OUDialog.alert(amountOfCurrency + " " + this.getCurrencyName()
        + " cost £" + df.format(cost));
}
```

Once you have got your `calculateCommission()` to compile correctly, you should write the method `displayCommission()`. Here is the method header:

```
public void displayCommission(int amountOfCurrency)
```

The method should determine the total commission payable on `amountOfCurrency`; it should do this by making use of a `calculateCommission()` message. The method should then display in a dialogue box the value returned by `calculateCommission()`. Note that to display the commission to two decimal places you will need to make use of the `DecimalFormat` class as shown in the code above. The message displayed should be something like the following:

To buy 60 euros will entail a commission cost of £4.07

Finally, once you have got your `displayCommission()` to compile correctly, modify the `costOfCurrency()` method so that it does not duplicate the code in your `calculateCommission()` method; instead `costOfCurrency()` should send a `calculateCommission()` message to this in order to get the cost of commission.

Once you have got your modified `costOfCurrency()` to compile correctly, test the new methods and the modified `costOfCurrency()` by executing the following code in the OUWorkspace:

```
Converter euroConverter = new Converter("euros", 1.45);
euroConverter.displayCommission(60);
euroConverter.costOfCurrency(60);
```

DISCUSSION OF ACTIVITY 7

A possible solution for the `calculateCommission()` method is given below, you may have written your method slightly differently.

```
public double calculateCommission(int amountOfCurrency)
{
    double percentageFee;
    percentageFee = Converter.getPercentageRate() * amountOfCurrency
        / this.getExchangeRate();
    return Converter.getFixedFee() + percentageFee;
}
```

Next, we have the `displayCommission()` method:

```
public void displayCommission(int amountOfCurrency)
{
    DecimalFormat df = new DecimalFormat("0.##");
    OUDialog.alert("To buy " + amountOfCurrency + " "
        + this.getCurrencyName()
        + " will entail a commission cost of £"
        + df.format(this.calculateCommission(amountOfCurrency)));
}
```

A modified `costOfCurrency()` method that makes use of `calculateCommission()` is shown below:

```
public void costOfCurrency(int amountOfCurrency)
{
    double commission, cost;
    DecimalFormat df = new DecimalFormat("0.##");
    commission = this.calculateCommission(amountOfCurrency);
    cost = commission + (amountOfCurrency / this.getExchangeRate());
    OUDialog.alert(amountOfCurrency + " " + this.getCurrencyName()
        + " cost £" + df.format(cost));
}
```

The statement:

```
euroConverter.displayCommission(60);
```

Should display the following message in a dialogue box:

To buy 60 euros will entail a commission cost of £4.07

The statement:

```
euroConverter.costOfCurrency(60);
```

Should display the following message in a dialogue box:

60 euros cost £45.45

If you have problems finishing the class Converter, the project Unit8_Project_8 contains the finished class.

3

Errors in programming

In this section we look at the various errors that can occur in code. There are two general types of computer programming error – compile-time errors (errors which occur when you try to compile a class) and run-time errors.

The compiler will report any compile-time errors it finds and usually supply some helpful information, such as the kind of error and the line number at which it occurs. Although removing all the compile-time errors may sometimes take a while, we can be sure the compiler will find them all, and with the assistance of the diagnostic details it provides we will sooner or later succeed in compiling our code. Only when this has happened is it possible to actually run the code.

The checking that the compiler carried out will have eliminated many errors in the program. However it will only have been able to deal with those that can be checked 'statically', i.e. without the program running. When execution takes place, run-time errors may occur. The program may not behave as intended or it may fail and stop completely. It is these errors that people mean when they talk about 'bugs'. Bugs can be very difficult indeed to correct, typically much harder than compile-time errors.

3.1 Compile-time errors

Compile-time errors can be subdivided into syntax errors and static semantic errors.

Syntax errors

The set of grammatical rules which govern how the elements of Java may be combined is called the **syntax** of the language. The term syntax error comes from linguistics.

In English 'The is sun a star' has a **syntax error**, 'is' cannot follow 'the' in a well-formed English sentence. In a similar way, we can say that an improperly formed Java sentence, i.e. an incorrectly formed Java statement, has a syntax error. In the first phase of compiling a class, the compiler in the BlueJ IDE will **parse** the code to determine its grammatical structure and this will reveal any syntax errors. Any such errors are reported in the message area of the BlueJ editor. Examples of common mistakes that give rise to syntax errors in Java are:

- ▶ omitting a semicolon at the end of a statement;
- ▶ omitting a bracket or brace;
- ▶ omitting a string delimiter (i.e. a double quote mark ");
- ▶ putting a Java keyword in an inappropriate part of the code;
- ▶ inserting a semicolon where there should not be one.

SAQ 4

Identify the syntax errors in the following statements.

- (a) HoverFrog h1 = HoverFrog() new;
- (b) String s1 = hello world";
- (c) int x = 55
- (d) int y = (10 / 2) * (3 + 4) + 6;

ANSWER.....

- (a) The new operator must come before a constructor, not after it.
- (b) There should be a double quote mark (string delimiter) before hello.
- (c) There should be a semicolon at the end of the statement.
- (d) There is a left-hand bracket missing in the compound expression. The expression should have been written as:

(10 / 2) * ((3 + 4) + 6);

or

((10 / 2) * (3 + 4) + 6);

depending on which order the programmer intended the sub-expressions to be evaluated.

SAQ 5

Identify the syntax error in the following hypothetical method for the class Frog.

```
/**  
 * Increment position of receiver by 1 and return the new position.  
 */  
public int moveRightAndGetPosition()  
{  
    return this.getPosition();  
    this.right();  
}
```

ANSWER.....

The `return` statement terminates the method and therefore the statement
`this.right();`

is unreachable. Unreachable statements are illegal in Java. More generally, a `return` statement can only appear as the last statement in a block.

We shall not study the syntax of Java in detail here. An informal knowledge of it is all that we shall require, although as the course progresses you will come to learn more.

Semantic errors

After verifying that your program is syntactically correct, the compiler's **parser** next checks for another kind of correctness – semantic correctness. Again, the term **semantics** comes from linguistics. In English, the sentence 'The sky is made of green cheese' is a **semantic error** – it is an error of meaning – the sky is certainly not made of green cheese. However, the sentence is syntactically correct; it has a correct form.

Examples of semantic errors would be.

- ▶ Using an undeclared variable in an expression.
- ▶ Using a variable in an expression before it has been given a value.
- ▶ Assigning a value of some type to a variable that has been declared of some other non-compatible type.
- ▶ Using an undefined identifier – this may occur because you have forgotten to declare a variable, or you may have mistyped a declared identifier (a variable name).

- Sending a message to an object that is referenced by a variable that has been declared as a type which does not support that message. For example:

```
Frog kermit = new HoverFrog();
kermit.up();
```

results in a semantic error. Although the HoverFrog object does have the message `up()` in its protocol, the variable `kermit` to which it is assigned, has been declared as being of type `Frog` which does not have `up()` in its protocol.

Suppose when typing in the code for method `costOfCurrency()`, in the class `Converter`, we inadvertently misspelt the message `calculateCommission()` as `calculateComission()` (with only one 'm') so that the method read as follows:

```
public void costOfCurrency(int amountOfCurrency)
{
    double commission, cost;
    commission = this.calculateComission(amountOfCurrency);
    cost = commission + (amountOfCurrency / this.getExchangeRate());
    OUDialog.alert(this.getCurrencyName() + amountOfCurrency
        + " cost £" + cost);
}
```

The entire method is syntactically correct, the 'grammar' of Java has been correctly followed and the message send:

```
this.calculateComission()
```

is also syntactically correct, however it is *semantically* incorrect, because it has no meaning in the context of a `Converter` object, because there is no message called `calculateComission()` in the instance protocol of the class `Converter` and the compiler would generate an error message.

SAQ 6

Why will the following code result in a semantic error?

```
Toad myToad = new Toad();
myToad.up();
```

ANSWER.....

The message `up()` is not in the protocol of `Toad` objects.

SAQ 7

Find the syntax and semantic errors in the following Java code.

- int x = 3 \$ 4
- int y = 3;


```
String s = y;
```
- int z = 4;


```
String t = "for 2";
      if (z == t) then z = 5;
```

ANSWER.....

- While the `$` letter can be used in identifiers, in its position in this line of code the compiler would interpret `$` as an operator, and there is no such operator in Java (syntax error).

Also there is no semicolon at the end of the line (syntax error).

- The second line of code tries to assign an `int` to a `String` (semantic error).

- (c) The condition `z == t` is comparing an `int` to a `String` (semantic error).

Also `then` is not a legal Java keyword but it is a legal identifier, so a compiler might think you were declaring a variable called `z` of type `then` and then assigning an `int` to it. So it is a syntax error or a semantic error depending on how you look at it!

In the next activity you will be asked to introduce some syntax and semantic errors into the **source code** for the class `Converter`, in order to understand the error reports that result from attempting to compile it. The ability to interpret the various error reports displayed by the BlueJ compiler is an important skill to learn. Throughout the activities, you will be experimenting with the code for the method `public void costOfCurrency(int amountOfCurrency)`.

ACTIVITY 8

Launch BlueJ and then open the project `Unit8_Project_8`. Double-click on the `Converter` class to open the editor.

Scroll down the code for the `Converter` class until you find the method header
`public void costOfCurrency(int amountOfCurrency)`.

- 1 Remove the closing parenthesis (round bracket) at the end of the fourth line of code:

```
cost = commission + (amountOfCurrency / this.getExchangeRate());
```

Then click the compile button and note the error message. Restore the closing parenthesis to the method and make sure that the class compiles correctly before proceeding.

- 2 Remove the opening parenthesis on the fourth line of code. Then click the compile button and note the error message:

```
cost = commission + (amountOfCurrency / this.getExchangeRate());
```

Restore the opening parenthesis to the method and make sure that the class compiles correctly before proceeding.

- 3 Remove the word `commission` and the comma from the variable declaration:

```
double commission, cost;
```

Then click the compile button and note the error message. Restore the word `commission` and the comma to the variable declaration and make sure that the class compiles correctly before proceeding.

- 4 Remove the argument `amountOfCurrency` from the third line of code:

```
commission = this.calculateCommission(amountOfCurrency);
```

Then click the compile button and note the error message. Restore the argument `amountOfCurrency` and make sure that the class compiles correctly before proceeding.

DISCUSSION OF ACTIVITY 8

- 1 The compiler should have displayed the error message `')' expected` and the line of code in which the `)` is missing should be highlighted.

The compiler's parser reads and analyses the code during compilation, checking that the code is syntactically correct (well formed). On encountering the opening parenthesis, it starts looking for the matching closing one. However before one is found, a semicolon indicating the end of the statement is encountered. So the parser displays the appropriate error message. This is an example of a syntax error.

- 2 The parser does not know that the programmer intended to use parentheses in this expression to calculate the cost. It does not detect an error until it has reached the unexpected closing parenthesis. Not unreasonably, it thinks that the code without parentheses was intended. It therefore assumes that the programmer meant to insert a semicolon rather than a bracket, so displays the message ';' expected and the line of code in which the error is found is highlighted. This is an example of a syntax error.

- 3 The parser displays the error message:

cannot find symbol - variable commission
and highlights the following line of code:

```
commission = this.calculateCommission(amountOfCurrency);
```

The parser is indicating that commission has not been declared as a variable. This is an example of a semantic error.

- 4 The parser displays the following error message:

calculateCommission(int) in Converter cannot be applied to ()
and highlights the following code:

```
commission = this.calculateCommission();
```

The parser sees that the method calculateCommission() needs an argument which is an int but that the message-send this.calculateCommission() does not include an argument. This is an example of a semantic error.

In the early stages of learning Java, you will probably spend a lot of time tracking down compile-time errors. As you gain experience, though, you will make fewer errors and find and correct them faster.

SAQ 8

What sorts of error are detected at compile time?

ANSWER.....

Syntax and static semantic errors are detected at compile time.

ACTIVITY 9

In this activity you are going to fix the compile-time errors in a new class called InterestCalculator. This class has a single static method calculateInterest() which calculates the compound interest of a fixed sum over a fixed number of years.

Launch BlueJ and open the project Unit8_Project_9 and then the InterestCalculator class. Try to compile the class. You will get a compile time error. Fix the error and then recompile. You will get another error.

- ▶ Work on the errors in order, and recompile after each error is fixed.
- ▶ Use the error messages and the highlighting of the code; they are there to help you.
- ▶ Sometimes the exact position at which the compiling process spotted an error is not the place where the mistake was made!
- ▶ Read the message and learn to recognise the technical language for different sorts of errors.
- ▶ Use your knowledge of Java to fix the error.

DISCUSSION OF ACTIVITY 9

The first error message you get is ';' expected. The following line is highlighted:

```
nestEgg = nestEgg + interest;
```

However the cause of the error is on the previous line where a semicolon is missing.

The second error is unclosed string literal and the following line is highlighted:

```
OUDialog.alert("After " + years + " years your nest egg would be worth £ + df.format(nestEgg));
```

The error is that there should be closing double quotes after the pound sign (£).

The first two errors were simple syntax errors, The next error is more interesting. The line:

```
rate = Double.parseDouble(OUDialog.request ("Enter interest rate"));
```

is highlighted and the error message cannot find symbol – variable rate is displayed. The problem is that rate is not declared, so it will not be recognised by the compiler as a variable. We need to add a declaration of rate as a double to the variable declarations at the top of the method.

So, is that it? The code is now fixed? No, we have only fixed the compile-time errors and there might be run-time errors to be fixed! We shall return to this code in Activity 12, in Section 4 which deals with debugging run-time errors.

3.2 Run-time errors

In general, compile-time errors are relatively easy to detect and are reported; translation and execution does not take place until they have been corrected. However compilers are not perfect and so do not catch all errors at compile time. Some errors only become apparent at run-time.

You may have heard about, or read reviews of, software which is described as having 'bugs'. The word **bug** seems to have entered the language of computing to mean any computer malfunction arising from the use of some piece of software. Its use is rather general, and describes run-time errors, and sometimes even hardware problems. Because of this rather imprecise usage we shall avoid the term, but it is now so common that you are bound to come across it, and may wonder how a word describing insects has become a generic term for a computer error.

Admiral Grace Hopper, the inventor of COBOL (Common Business Oriented Language), made famous the story of how a computer technician solved a problem with an early computer by removing an insect from inside it. This led to her describing any general computer problem as a bug – the common use of the word today. However, the use of the word was soon reserved for software errors.

Run-time errors can be subdivided into logical errors and dynamic semantic errors (sometimes called execution errors).

Logical errors

A **logical error** is the result of code not correctly implementing the specification of a particular problem. Although the code is both syntactically and semantically correct, it does not behave as expected at run-time. For example, the wrong results may be returned. As a trivial example, when writing a method, you may have got muddled

between left and right, and sent a Frog object the message `right()` when you wanted it to move left in the microworld.

Suppose in the currency converter example we had not fully understood how the components of the commission charges were calculated. This would lead to an incorrect equation for the calculation of the final cost, and hence to a code solution which, although syntactically and semantically correct, returned an incorrect calculation.

Suppose we thought that the equation for calculating the commission when buying some foreign currency was:

$$\text{commission} = \frac{\text{fixed fee} + (\text{percentage rate} * \text{amount of currency})}{\text{exchange rate}}$$

instead of the correct equation

$$\text{commission} = \frac{\text{fixed fee} + (\text{percentage rate} * \text{amount of currency})}{\text{exchange rate}}$$

Then we would implement `calculateCommission()` as follows.

```
public double calculateCommission(int amountOfCurrency)
{
    return (Converter.getFixedFee() + (Converter.getPercentageRate()
        * amountOfCurrency)) / this.getExchangeRate();
}
```

This is syntactically correct but implements the incorrect equation. If the incorrect equation and the following code were used and executed:

```
Converter dollarConverter = new Converter();
dollarConverter.setCurrencyName("US dollars");
dollarConverter.setExchangeRate(1.5);
```

The statement:

```
dollarConverter.costOfCurrency(100);
```

would result in a dialogue box displaying:

US dollars 100 cost £71.33

rather than

US dollars 100 cost £71.99

Such an error could go undetected for some time, costing Walton Bureau de Change, over time, a great deal of money. It is the formula not the coding that is at fault in that a logical error was made in the algorithm for solving the original problem. The code is correctly performing what it was asked to do. It might be quite tricky for a programmer to determine the cause of the incorrect answer because, while it is the message-send `dollarConverter.costOfCurrency(100)` that has produced a wrong answer, the error is in the method `calculateCommission()`.

A logical error like this can be difficult to detect because neither the compiler nor the Java Virtual Machine (JVM) can give us any help. As far as they are concerned all is well – the syntax rules and semantics of the Java language have been obeyed and the method computes the expressed logic. Neither the compiler nor the Java Virtual Machine (JVM) can read the programmer's mind!

Imagine how such a logical error made in formulas for trading stocks and shares could quickly multiply into a multimillion-pound mistake.

One way of detecting logical errors is to generate data against which the computed results can be compared. This often involves working out values by hand and comparing them with values resulting from message executions. The process is called testing and while testing cannot guarantee that you have the correct logic, it often reveals the presence of errors in code. Here, because the calculations involved are simple numerical ones, testing would be relatively easy to carry out; it would involve doing some calculations by hand and comparing them with the computed values. Since we could not verify every possible conversion, the difficulty would be in deciding which values would best test the code and give us some confidence that it was correct. We shall not attempt to answer that now but will return to testing in *Unit 13*.

SAQ 9

What do you understand by run-time?

ANSWER.....

Run-time is when the bytecode produced by the compiler is executed on the JVM.

SAQ 10

A Frog object aFrog and a Toad object aToad are to be included in a method. At the end of the method, these objects are to be positioned on the leftmost 'stone'. The programmer writes the following code at the end of the method.

```
aFrog.home();  
aToad.home();
```

What type of error has the programmer made? Give the correct code.

ANSWER.....

A logical error has been made. The 'home' position for a Toad object is the rightmost stone. The code should be as follows.

```
aFrog.home();  
aToad.setPosition(1);
```

Dynamic semantic errors

Another type of runtime error is a **dynamic semantic error**, dynamic because it is a semantic error that cannot be detected by the compiler, but which *can* be detected by the JVM or a method at runtime (if it has been written to do so). If a method has been written to detect a particular kind of dynamic semantic error, and it does encounter such an error, it is said to **throw** an exception. An **exception**, is so called because it usually indicates that something exceptional (and usually bad) has happened: a condition has occurred in the executing method which means the normal flow of execution cannot continue safely. The exception that is thrown is actually a Java object that contains information about the error, including its type and the state of the program when the error occurred. It is thrown with the hope that another method, which sees the bigger picture and so knows better how to resolve the problem, will **catch** it and will either take action allowing the program to continue, or at least arrange for the program to terminate gracefully. The actions that are taken as a result of an exception being thrown are called **exception handling**.

When a method throws an exception it is first caught by the JVM which then attempts to find another method to handle that exception. The set of possible methods that could handle the exception is the ordered list of methods that have been called (invoked) to get to the method where the error occurred. This list of methods is known as the **call stack** – Figure 1 shows a snapshot of the call stack after a `doTransactions()` message has been sent to a `Converter` object and shows the state of the call stack once the method `calculateCommission()` has invoked the `getPercentageRate()` method on the `Converter` class.

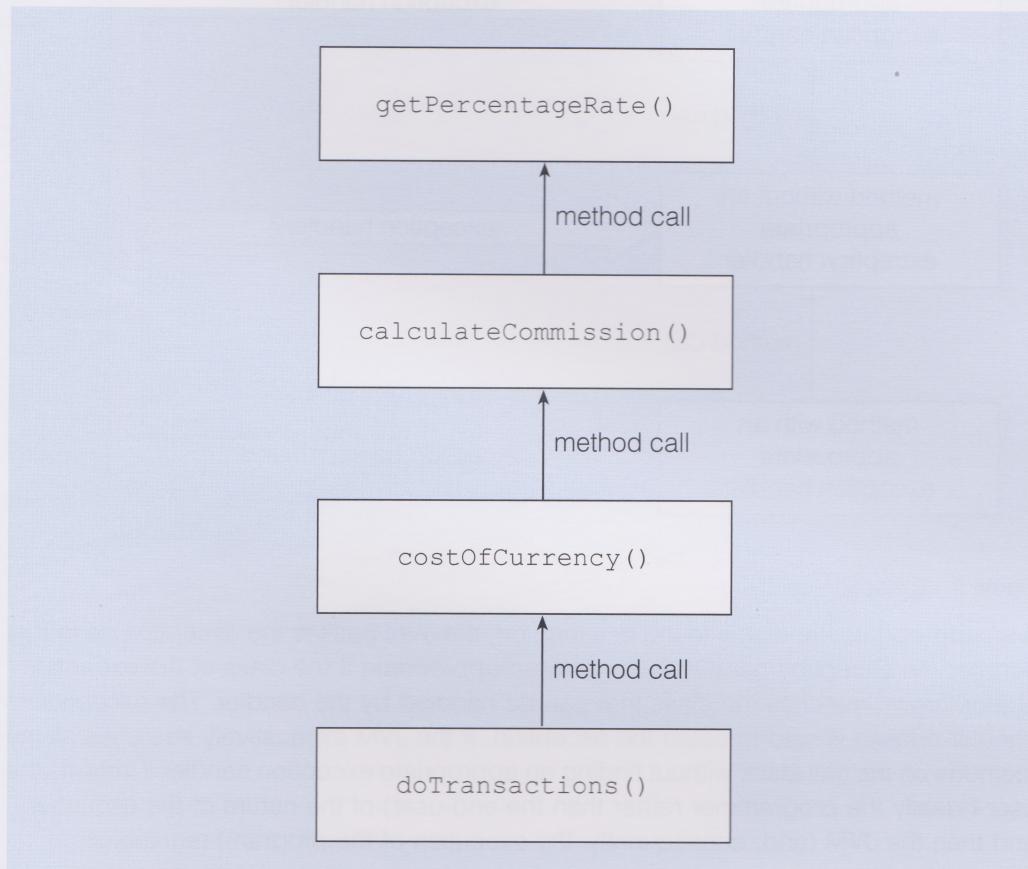


Figure 1 Snapshot of the call stack after a `doTransactions()` message has been sent to a `Converter` object

If the method on the top of the call stack (the currently executing method) throws an exception, the JVM searches down the call stack for a method that contains a block of code that can catch that exception. This block of code is called an **exception handler**. The search begins with the method that called the method in which the error occurred, and proceeds down through the call stack (as depicted by Figure 2).

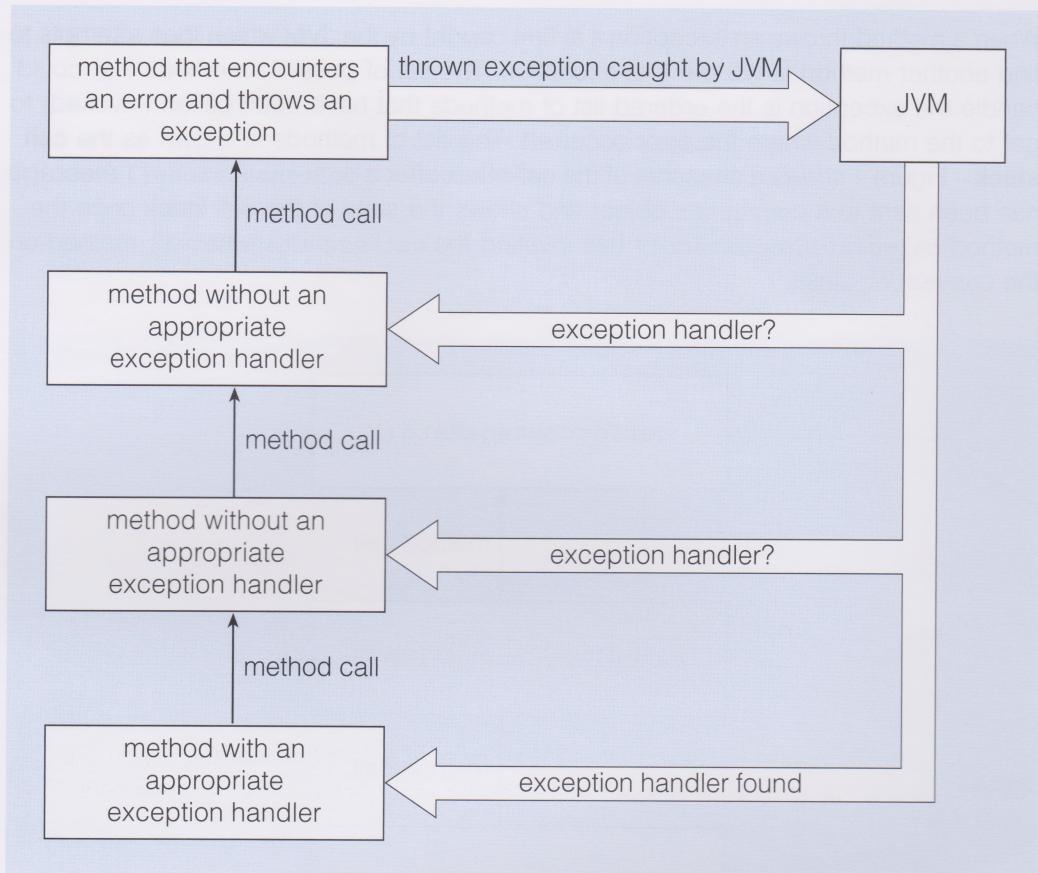


Figure 2 Exception catching

If an appropriate handler is found in a method, the JVM passes the exception on to that handler. An exception handler is considered appropriate if the class of the exception object thrown matches the class that can be handled by the handler. The exception handler chosen is said to catch the exception. If the JVM exhaustively searches all the methods on the call stack without finding an appropriate exception handler it informs the user (ideally the programmer rather than the end-user) of the nature of the exception and then the JVM (and, consequently, the execution of the program) terminates.

The most common exceptions that you are likely to have encountered so far include:

- ▶ `ArithmaticException` – an instance of this class is thrown when an exceptional arithmetic condition has occurred; for example, when division by zero is required.
- ▶ `NullPointerException` – an instance of this class is thrown when you try and access an instance variable of, or send a message to, a reference variable that is set to null as it does not yet reference an object.
- ▶ `NumberFormatException` – an instance of this class is thrown to indicate that a method that converts a string to one of the numeric types has been passed a string, which does not have the appropriate format, as its argument.

The exception classes above are all built-in exceptions: they are part of the standard Java library. In addition to these built-in exceptions, Java provides a mechanism for the programmer to define other types of exceptions but this is beyond the scope of this course.

An example of a method that throws an exception is the `Integer` class's static method `parseInt()` which you used in Activity 6 for the `doTransactions()` method. Here is the code for the method:

```
public void doTransactions()
{
    String amountString;
    int amountOfCurrency;
    boolean repeat = true;
    while (repeat)
    {
        amountString = OUDialog.request("How many " + this.getCurrencyName() + "?");
        amountOfCurrency = Integer.parseInt(amountString);
        this.costOfCurrency(amountOfCurrency);
        repeat = OUDialog.confirm("Would you like another transaction?");
    }
}
```

Sending the `doTransactions()` message to a `Converter` object would result in the `doTransactions()` method starting to execute. Within the `while` loop of the method, the `request()` method is invoked on the `OUDialog` class to display a dialogue box. If in response to this request dialogue box, the user entered `fifty` rather than `50`, the local variable `amountString` would hold the string `"fifty"`. Next the method invokes the `parseInt()` method on the `Integer` class with `amountString` as the argument. This would give us a call stack as depicted in Figure 3.

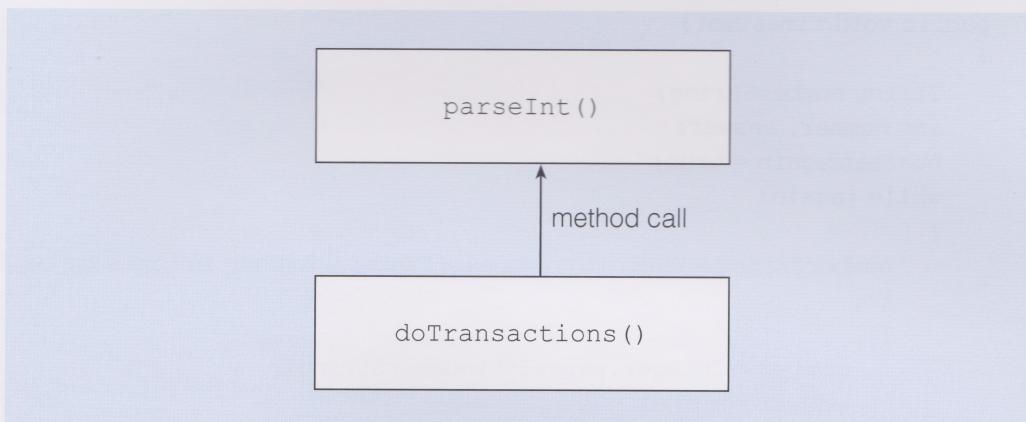


Figure 3 Call stack after `doTransactions()` has invoked `parseInt()`.

As the `parseInt()` method starts to execute it would throw an instance of the `NumberFormatException` class because it was expecting, as its argument, a string that contained only digits, rather than a string containing alphabetic characters ("fifty"). The exception would be caught by the JVM which would then look for an appropriate event handler. As there is only one other method on the call stack, `doTransactions()`, the JVM has only one place to look. As our code for `doTransactions()` does not attempt to catch instances of the `NumberFormatException` class (or indeed any other class of exception), the JVM just informs the user that a `NumberFormatException` has occurred and terminates execution.

This is not an ideal state of affairs; it would seem that our code is not very robust! However Java does give us a mechanism to catch exceptions – the try–catch statement. Here is the syntax:

```
try
{
    < try block statements >
}
catch(ExceptionClass e)
{
    < catch block statements >
}
```

The try block is prefixed by the keyword try and it is here (between the curly brackets) that we put code that we suspect might result in an exception. The catch block is prefixed by the keyword catch and then in parenthesis there is an argument declaration just like the argument declaration for a method. This argument declaration gives the class of the argument (in this case a made-up class name, ExceptionClass, for the purpose of showing the syntax) and the name of the argument, in this case e. The only exceptions thrown by the try block that the catch block can catch are those which match the class of the catch block's argument. So the catch block will only ever execute if the try block throws an exception which matches the class of the catch block's argument declaration.

Let us see what this would look like in the context of a hypothetical method called timesTwo().

```
public void timesTwo()
{
    String numberString;
    int number, answer;
    boolean again = true;
    while (again)
    {
        numberString = OUDialog.request("Enter a number to times by two");
        try
        {
            number = Integer.parseInt(numberString);
            answer = number * 2;
            OUDialog.alert(numberString + " * 2 = " + answer);
        }
        catch(NumberFormatException anException)
        {
            OUDialog.alert("The string entered did not contain an integer");
        }
        again = OUDialog.confirm("Would you like another go?");
    }
}
```

In the above code, if parseInt() throws an exception, execution of the timesTwo() method will jump immediately to the catch block and the other two lines in the try block will not be executed. If no exception is thrown the catch block is not executed. The code after the catch block is always executed. Notice how a catch takes an argument, just like a method. The argument in this case is called anException and it is declared to be of class NumberFormatException. This defines that the catch block will only handle cases of NumberFormatException, it will not handle any other class of exception.

ACTIVITY 10

- 1 If it is not already open, launch BlueJ and open the project Unit8_Project_10. This project contains the class `Converter` from the previous section. Next open the OUWorkspace and execute the following code:

```
Converter dollarConverter = new Converter("US dollars", 1.5);
dollarConverter.doTransactions();
```

When the `doTransactions()` method requests input, try entering into the text box of the request dialogue box the word `fifty` and observe how the method behaves. Observe any output in the Display Pane.

- 2 Next, in the BlueJ window, double-click on the icon for the `Converter` class to open the editor and then add code to the `doTransactions()` method so that it catches any `NumberFormatException` thrown by `parseInt()` (just as was shown in the `timesTwo()` method above). Once you have done that, and you have successfully recompiled the class, try the following code once more in the OUWorkspace:

```
Converter dollarConverter = new Converter("US dollars", 1.5);
dollarConverter.doTransactions();
```

As the code in the `doTransactions()` method loops, try entering into the request dialogue box `50`, `fifty`, `twenty` and `20` in turn and once again observe how the method behaves.

DISCUSSION OF ACTIVITY 10

- 1 Entering `fifty` into the request dialogue box results in the following message being displayed in the Display Pane (if both lines are executed together).

```
Exception: line 2. java.lang.NumberFormatException:
For input string: "fifty"
```

- 2 Here is the code you should have written.

```
public void doTransactions()
{
    String amountString;
    int amountOfCurrency;
    boolean repeat = true;
    while (repeat)
    {
        amountString = OUDialog.request("How many "
            + this.getCurrencyName() + "?");
        try
        {
            amountOfCurrency = Integer.parseInt(amountString);
            this.costOfCurrency(amountOfCurrency);
        }
        catch(NumberFormatException anException)
        {
            OUDialog.alert("The string entered did not represent an integer");
        }
        repeat = OUDialog.confirm("Would you like another transaction?");
    }
}
```

You should have observed that entering fifty or twenty into the request dialogue box no longer caused execution of the method to cease and an error message to be displayed in the Display Pane, instead the statement in the catch block is executed and the method continues executing.

If you had problems with this activity, the solution code has been added to Unit8_Project_11.

Exceptions are objects so we can send messages to them. For example, we could have written the catch statement in the previous activity as:

```
catch(NumberFormatException anException)
{
    OUDialog.alert(anException.toString());
}
```

In this code, the message `toString()` is sent to the exception object to get its textual representation which in this case is displayed in a dialogue box. When `toString()` is sent to a `NumberFormatException` object the message answer looks like the following:

`java.lang.NumberFormatException: For input string <some string>`

where `<some string>` is the string that caused the exception. You will notice that this is the same string that was shown in the Display Pane in part 1 of Activity 10, before we put the try-catch statement into our code.

ACTIVITY 11

If it is not already open, launch BlueJ and open the project Unit8_Project_11. This project contains the `Converter` class with the changes made in Activity 10.

In the BlueJ window, double-click on the `Converter` icon to open the editor and then in the method `doTransactions()` change the code in the catch block so that a dialogue box shows the result of sending `toString()` to the caught exception. Once you have done this and successfully recompiled the `Converter` class, open the OUWorkspace and try the following code once more in the OUWorkspace:

```
Converter dollarConverter = new Converter("US dollars", 1.5);
dollarConverter.doTransactions();
```

When the `doTransactions()` method requests input, try entering into the request dialogue box fifty and observe how the method behaves.

DISCUSSION OF ACTIVITY 11

The code in the catch block should now look like the following:

```
catch(NumberFormatException anException)
{
    OUDialog.alert(anException.toString());
}
```

Entering fifty into the request dialogue box now results in an alert dialogue box displaying the following string:

`java.lang.NumberFormatException: For input string: "fifty"`

If you had problems with this activity, the solution is in Unit8_Project_11_sol.

In Java there are two families of exceptions: **checked exceptions** and **unchecked exceptions**. Checked exceptions are subclasses of the class `Exception`, and unchecked exceptions are subclasses of the class `RuntimeException`. There is an important difference between how Java handles checked and unchecked exceptions.

- ▶ If you write a method that causes the execution of a method that throws a *checked* exception, you must try and handle that exception. It is enforced by the compiler; your code will not compile until you explicitly either try and catch that exception or alternatively give your method a signature that declares it also throws that exception, so making the code that invokes your method deal with it.
- ▶ If you write a method that causes the execution of a method that throws an *unchecked* exception, the compiler does not force you to handle that exception, although you as the programmer may decide to catch that exception – as you did in the previous activity.

The exceptions we have mentioned in this section are all unchecked exceptions. In *Unit 12* you will be introduced to checked exceptions.

4 Debugging

Debugging is the process of removing run-time errors from programs. Typically at this stage, the program compiles and runs, but one or more errors prevent the program from functioning properly.

In an ideal world programs would be written correctly the first time and therefore never require debugging. Almost as ideal is catching the errors in the program simply by thoroughly examining it, usually mentally running it on various test cases to verify what should be true at this point. Evaluating the correctness of a program by examining the code is known as inspection. If the program appeared to be correct on inspection, but errors become apparent as it runs, how should we set about tracking down the causes?

Probably the most common technique that programmers employ to debug code is to add output statements to a program. These statements are strategically placed in the code to inform the programmer of the flow of control and the values of key variables. The output produced may make the problem obvious or may be used to successively narrow down the problem location.

In Java there is a class called `System` that is always in scope (it is automatically imported into every Java program). This class gives the programmer access to system resources, one of these resources is the output console, `out`, which is an object which will send text to a particular window or pane. In the context of the OUWorkspace `out` will display text in the Display Pane if you send it a `println()` message. For example, typing and executing the code:

```
System.out.println("I'm here");
```

will display the following in the Display Pane of the OUWorkspace:

```
I'm here
```

4.1 Debugging if, for and while statements

In this section we look at how you can use `println()` statements to debug `if`, `for` and `while` statements.

if statements

When debugging code that involves an `if` statement, it is often useful to find out which branch is executing. You can do this by putting `println()` statements at the top of each branch:

```
if (test)
{
    System.out.println("test was true so executing then block");
    // rest of the then block statements
}
else
{
    System.out.println("test was false so executing else block");
    // rest of the else block statements
}
```

In the aforementioned code if the condition `test` evaluates to `true`, the code will print out that the `then` block is executing, if `test` evaluates to `false`, the code will print out that the `else` block is executing.

while statements

When debugging `while` statements, we are generally interested in the value of the condition that controls whether the `while` block executes or not. If the condition evaluates to `false` the first time through the loop, the statements inside the block are never executed, so it is useful to output the value of the condition just before the `while` statement. So long as the condition evaluates to `true` before the `while` statement then the `while` block will execute at least once. The code in the block should change the value of one or more variables so that, eventually, the condition evaluates to `false` and the loop terminates, otherwise the loop would repeat for ever (called an infinite loop). Therefore the last statement in the block should again output the value of the condition so that you can determine that the condition eventually evaluates to `false`.

```
System.out.println("Before while statement: test = " + test);
while (test)
{
    //while block statements
    System.out.println("Inside while block: test = " + test);
}
```

for statements

As you know, `for` statements are used when we want a block of code to execute a fixed number of times. For example, the following

```
for (int controlVar = 1; controlVar < 3; controlVar = controlVar + 1)
{
    //for block statements
}
```

would execute twice, whereas

```
for (int controlVar = 1; controlVar <= 3; controlVar = controlVar + 1)
{
    //for block statements
}
```

would execute exactly three times. The difference between these two `for` statements is that the first one has the condition `controlVar < 3`, whereas the second has the condition `controlVar <= 3`. It is relatively easy to get these Boolean conditions wrong, so to make sure that the `for` block executes the correct number of times, you can output the value of the control variable as the first line of the block, for example:

```
for (int controlVar = 1; controlVar <= 3; controlVar = controlVar + 1)
{
    System.out.println("The value of controlVar is " + controlVar);
    //rest of the for block statements
}
```

When the above executes, the following output would be displayed in the Display Pane of the OUWorkspace:

```
The value of controlVar is 1
The value of controlVar is 2
The value of controlVar is 3
```

confirming that the `for` block is executed three times.

ACTIVITY 12

The static method `calculateInterest()` in the class `InterestCalculator` has a logical error; no matter what integer is entered for the number of years to calculate the interest, the calculation seems to only calculate years -1. Since it is within the `for` loop that this calculation takes place, it is sensible to put a `println()` message within the `for` loop to print out the value of the control loop variable `year`. Open `Unit8_Project_12` and add a line of code to the `for` loop so that it includes a `println()` message to display the value of the control loop variable as indicated below in bold.

```
for (int year = 1; year < years; year = year + 1)
{
    System.out.println("year is now " + year);
    interest = nestEgg * rate / 100.0;
    nestEgg = nestEgg + interest;
}
```

Compile the code and then, in the OUWorkspace, send the message `calculateInterest()` to the `InterestCalculator` class and, when prompted for the number of years to invest, enter the number 5.

DISCUSSION OF ACTIVITY 12

In the OUWorkspace you should have evaluated the following code:

```
InterestCalculator.calculateInterest();
```

After prompting you for the interest rate, the sum to be invested and the number of years to invest, your code should print the following to the Display Pane of the OUWorkspace:

```
year is now 1
year is now 2
year is now 3
year is now 4
```

It is clear from the output that the `for` statement block never executes for year 5 – what is wrong? Well, let us take a closer look at the condition for the loop which is `year < years`.

The first time the condition is tested `year = 1` which is less than `years` which holds the value 5, so the statement block is executed and the interest is calculated.

The second time the condition is tested `year = 2` which is less than `years`, so the statement block is executed and the interest is calculated.

The third time the condition is tested `year = 3` which is less than `years`, so the statement block is executed and the interest is calculated.

The fourth time the condition is tested `year = 4` which is less than `years`, so the statement block is executed and the interest is calculated.

Now the fifth time the condition is tested `year = 5` which is *not* less than `years`, so the statement block *is not* executed and the interest for that year is not calculated.

The error is clearly in the `for` loop's condition, so let us look at that condition again:

```
year < years
```

We know that a `for` loop will only exit when the Boolean condition evaluates to `false`. With `years` holding 5, this will happen when `year` gets to 5. When that happens, the `for` loop will terminate. One way to avoid this and to get the loop to execute a correct number of times is to add 1 to `years` in the condition:

```
year < years + 1.
```

This would work, but is poor programming style because it makes the code more difficult to read. What we want to make explicit in the code is that if `years` is 5, the loop should execute five times. The real problem is that we are using the wrong operator. In the condition we are testing whether `year` is *less than* `years`, when what we really should be doing is testing whether `year` is *less than or equal to* `years`. So the correct fix would be to change the condition of the `for` loop to read `year <= years`. Using the wrong comparison operator is easily done and a potent source of errors.

4.2 Tracing execution

In the previous subsection you saw how `println()` messages can be used to ascertain how many times a `for` loop's statement block is executed, and which branch of an `if-then-else` statement is executed. However `println()` can also be used in your code to ascertain what methods are executing at a particular time and in what order.

When you send a message to an object it often results in that object sending another message to either itself or another object. Often it can be hard to tell what other messages will be sent because it may depend on the outcome of a conditional statement. In such circumstances it can be hard to track down logical errors in your code. To facilitate this it is often useful to add `println()` messages in your methods so that when a method is executed it prints out some or all of the following information:

- ▶ the method name;
- ▶ the state of the receiver;
- ▶ the values of local variables and arguments.

The uses of `println()` statements are not restricted to dealing with error conditions. They can also be used to help in understanding error-free code by tracing the execution of the code message by message. We shall look at tracing error-free code in this subsection.

We look at a very simple class called `Debug` that has been designed to illustrate how to trace through code. It should not be treated as an example of good programming!

Class Debug

The class Debug does not have any instance variables. It has three instance methods `alpha()`, `beta()` and `gamma()` the code for which is given below.

```

/**
 * Sends the receiver the message beta()
 */
public void alpha()
{
    int answer;
    answer = this.beta();
}

/**
 * Sends the receiver the message gamma()
 */
public int beta()
{
    return this.gamma();
}

/**
 * Returns the number 42
 */
public int gamma()
{
    return 42;
}

```

When an instance of `Debug` receives the message `alpha()` the corresponding method is found and the `alpha()` method starts to execute. This starts a chain of further messages (`beta()`, then `gamma()`), as the method `alpha()` sends the message `beta()` and the method `beta()` sends the message `gamma()`.

At this point three messages have been sent in the order of `alpha()`, `beta()`, `gamma()`, but none of the corresponding methods have completed execution. The method `gamma()` then returns the number 42 – and so completes its execution. The method `beta()` can now complete, returning the result of `this.gamma()` to the method `alpha()`, and then `alpha()` can similarly complete execution. So a message answer is returned first by `gamma()`, then by `beta()`; this is the reverse of the order in which the messages were sent.

ACTIVITY 13

Launch BlueJ and then open the project `Unit8_Project_13`. Double-click on the class `Debug` to open the editor. Your task is now to add `println()` statements to the methods.

In the method `alpha()` add a line of code just after the declaration of the local variable `answer` that will print out "Hello from alpha()!". Then add a final line of code that will print out the value of the local variable `answer` using `println()`. The `println()` method is overloaded so its argument can be a string or a number so it is quite legal to use `answer` as the argument to `println()`. However your output will be more meaningful if you concatenate a suitable string such as "The value of `answer` is " with the variable `answer`.

In the method `beta()` add a first line of code that will print out "Hello from beta()!".

In the method `gamma()` add a first line of code that will print out "Hello from gamma()!".

Once you have edited the methods and successfully recompiled the class, open the OUWorkspace and execute the following code, observing what is displayed in the Display Pane:

```
Debug d = new Debug();
d.alpha();
```

DISCUSSION OF ACTIVITY 13

The code for your methods should now look like this:

```
/**
 * Sends the receiver the message beta()
 */
public void alpha()
{
    int answer;
    System.out.println("Hello from alpha()!");
    answer = this.beta();
    System.out.println("The value of answer is " + answer);
}

/**
 * Sends the receiver the message gamma()
 */
public int beta()
{
    System.out.println("Hello from beta()!");
    return this.gamma();
}

/**
 * Returns the number 42
 */
public int gamma()
{
    System.out.println("Hello from gamma()!");
    return 42;
}
```

After executing `d.alpha()` in the OUWorkspace the following should be displayed in the Display Pane:

```
Hello from alpha()!
Hello from beta()!
Hello from gamma()!
```

The value of answer is 42

Using carefully placed `println()` messages is a standard way of enabling tracing or debugging of code.

The project Unit8_Project_13_sol contains a version of the Debug class with the `println()` statements added.

ACTIVITY 14

In this activity you will experiment a little more on tracing program flow using `println()` statements.

Launch BlueJ and then open the project Unit8_Project_14. Double-click on the class Maze to open the editor.

The class declares the following variables:

```
private String s1 = "nine";
private String s2 = "time";
private String s3 = "A stitch";
private String s4 = "saves ";
private String s5 = "in";
```

We have partially completed three instance methods for you. Your task in this activity is to complete the methods by adding `println()` statements to these methods, where indicated by comments, that make use of the above instance variables. The hard bit of this activity is that you should use the above instance variables in the `println()` statements in such a way so that when the following code:

```
Maze mz = new Maze();
mz.m1();
```

is executed in the OUWorkspace, the Display Pane displays the well-known saying:

```
A stitch
in
time
saves
nine
```

DISCUSSION OF ACTIVITY 14

Here is the code you should have written showing you the order in which the `println()` statements are executed.

```
public void m1()
{
    System.out.println(s3); // executes first
    this.m3();
    System.out.println(s1); // executes fifth
}

public void m2()
{
    System.out.println(s2); // executes third
}

public void m3()
{
    System.out.println(s5); // executes second
    this.m2();
    System.out.println(s4); // executes fourth
}
```

5

Summary

After studying this chapter you should understand the following ideas.

- ▶ It is important to format code according to a set of widely followed guidelines so that it is readable and understandable by yourself and by others.
- ▶ Compile-time errors occur due to mistakes in the syntax or semantics of the source code. The compiler detects these errors and provides error messages to help the programmer to fix the errors.
- ▶ Run-time errors occur because of flaws in the programmer's logic or a misunderstanding of the syntax or semantics used.
- ▶ Java provides a powerful tool for dealing with the unpredictable kind of run-time errors. When an irregular situation occurs in a running program, the Java Virtual Machine creates an `Exception` object, which is passed to whichever object can best sort out the problem.
- ▶ Debugging is a process of ridding a program of avoidable run-time errors. Using `println()` messages to display information about the running program can aid in debugging.

LEARNING OUTCOMES

After studying this unit you should be able to:

- ▶ format Java code in accordance with the guidelines given in this unit;
- ▶ understand what is meant by compile-time errors and explain the difference between syntax and semantic errors;
- ▶ use the errors displayed by the compiler to track down and fix compile-time errors;
- ▶ understand what is meant by a run-time error and explain the difference between logical and dynamic semantic run-time errors;
- ▶ write simple code to catch an exception;
- ▶ use the debugging techniques taught in this unit to fix run-time errors.

Glossary

bug The cause of a **run-time error**.

catch The process of catching an **exception** – see try–catch statement.

checked exception An **exception** that the compiler requires the programmer to handle if they write code that executes a method that can throw such an **exception**.

compiler Software which checks that text written in a high-level language is correctly formed and, as far as can be determined before compilation, that it is meaningful **source code** for the language. If the check is successful, then the source code is translated into machine code. The particular machine code is generated for execution by the hardware of a real computer, or for some languages, such as Java, execution 'on' a virtual machine.

compile-time error Errors that are to do with the form of the text as determined by the rules for a given programming language. These are detected and reported during the early stages of compilation (see **compiler**). The compilation does not proceed to code generation if an error is detected.

debugging The identification and removal of **run-time errors** (bugs) from a program.

dynamic semantic error A **semantic error** that cannot be detected by the **compiler**, but which *can* be detected by the JVM or a method at **run-time** and which results in an exception being thrown.

exception An object that is thrown by a method or the JVM as the result of a **run-time error**. The exception object holds details of what went wrong allowing the exception handler which catches the exception to take appropriate action.

exception handling The programmed catching of an **exception** and the subsequent execution of reliable code to abandon execution, for example, or to restore the software to a meaningful state.

formatting guidelines A set of guidelines which specify how program code should be laid out.

logical error The result of code not correctly implementing the specification of particular problem. Although the code is both syntactically and semantically correct, it does not behave as expect at **run-time**.

parse See **parsing**.

parser That part of the **compiler** which checks **source code** for syntactic and semantic correctness.

parsing The process of deciding whether the input text is a 'sentence' of a given language and obeys its syntax and semantic rules.

run-time Refers to the moment when a program begins to execute, in contrast to the time at which it has been loaded or compiled. The amount of time, elapsed time, used in executing a program is sometimes called the run-time.

run-time error A programming error that only becomes apparent when the program is run, and cannot be detected beforehand. Run-time errors can be **logical errors** or **exceptions**.

run-time system The Java run-time system consists of the virtual machine plus additional software, such as class libraries, that are needed to implement the Java API on your operating system and hardware.

scope The scope of a variable describes the areas of program code from which the variable may be used. The scope of a local variable is the statement block in which it is declared (and any nested statement block). In discussing methods, the scope of a local variable is the method in which it is declared.

semantics That part of the definition of a language concerned with specifying the meaning or effect of text that is constructed according the syntax rules of the language.

semantic error A semantic error arises from a misunderstanding of the meaning or effect of some construct in a programming language. Many such errors are detected during compilation by the **parser**.

source code Text expressed in a high-level programming language. The term is also often applied to text that does not fully conform to the language (it contains an error), but with minor correction would conform.

syntax Rules defining the legal sequences of characters in a programming language. Syntax rules define the form of the various constructs in the language, but say nothing about the meaning of these constructs.

syntax error Failure to observe a syntax rule. Such an error is detected during compilation by the **parser**.

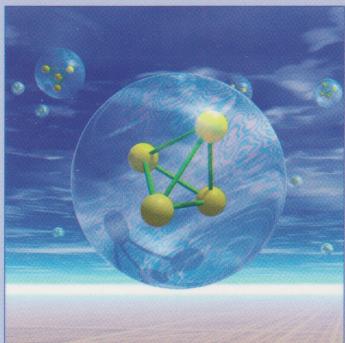
throw the process of throwing an exception, see **exception**.

try-catch statement An exception handler, a mechanism to catch **exceptions**.

unchecked exception An **exception** that the compiler *does not* require the programmer to handle if they write code that executes a method that can throw such an exception.

Index

B	E	S
bug 32	exception 34	scope 21
C	handler 35	semantic error 28
call stack 35	handling 34	syntax error 27
catch 34	F	T
checked exception 41	formatting guidelines 6	throw 34
compile-time errors 27	L	try-catch statement 38
D	logical error 32	U
debugging 42	P	unchecked exception 41
dynamic semantic error 34	parse 27	
	R	
	run-time error 32	



Block 2

- Unit 5 Dialogue boxes, selection and iteration
- Unit 6 Subclassing and inheritance
- Unit 7 Code design and class members
- Unit 8 Designing code, dealing with errors

